Co-simulation of Embedded Systems in a Heterogeneous MoC-Based Modeling Framework

Seyed Hosein Attarzadeh Niaki*, Ingo Sander*
*Electronic Systems
School of Information and Communication Technology
KTH Royal Institute of Technology, Stockholm–Sweden
Email: {shan2,ingo}@kth.se

***Abstract*—New design methodologies and modeling frameworks are required to provide a solution for integrating legacy code and IP models in order to be accepted in the industry. To tackle this problem, we introduce the concept of wrappers in the context of a formal heterogeneous embedded system modeling framework. The formalism is based on the language-independent concept of models of computation. Wrappers enable the framework to co-simulate/co-execute with external models which might be legacy code, an IP block, or an implementation of a partially refined system. They are defined formally in order to keep the analyzability of the original framework and also enable automations such as generation of model wrappers and co-simulation interfaces. As a proof of concept, three wrappers for models in different abstraction levels are introduced and implemented for two case studies.**

I. Introduction

Embedded systems are becoming increasingly more complex in terms of size and functionality. Designers need to cope with the increasing demand to integrate more components and IP-Blocks in a single design. A formal heterogeneous modeling framework addresses this problem in two ways. First, its formalism allows application of formal methods such as verification, performance analysis, refinement and synthesis in the design flow. Second, its support for heterogeneity allows systems with various components such as software or hardware, digital or analog, etc. to be specified, verified, and implemented all together in a single framework.

However, legacy code integration is one of the main challenges faced by every new design methodology and modeling style. Industrial companies would like to gain the benefits of new methods while still being able to reuse their already developed IP blocks. Legacy code and models are not usually compatible with the new paradigm and need their own simulation/execution environment. In addition, in an industrial top-down design approach, several tools and languages are used to implement different components of the system. In this case, verifying a refined system needs co-simulation of various tools together.

As a consequence, a formal heterogeneous modeling framework that can be used in an industrial setting, needs to provide a solution to integrate different external components in the framework without leaving the benefits of formalism for the rest of the system. In order to verify the functionality of the integrated system, a heterogeneous co-simulation is also needed to be performed.

In this work, we will try to develop a method for heterogeneous co-simulation of different tools and languages. The heterogeneous formal system specification framework *ForSyDe* [1] has been chosen and extended to be able to interact with external models and tools using the concept of *wrappers*. This is done by adding constructs with precise semantics to the original framework and templates for top-level wrappers of the codes being simulated externally. The framework acts as the driving engine for the co-simulation and orchestrates the simulation tools by synchronizing them and communicating input and output data with them.

The modeling framework already supports several computational models and their interaction via domain interfaces. Heterogeneity can also reveal itself not in having distinct computational models, but in having sub-parts of the system modeled in different languages. This could be because we might have both hardware and software components in our final design, and perhaps we need to represent our system at different abstraction levels in each step of the design flow. This latter case is contribution of the present work.

The contributions presented in this paper can be summarized as:

- adding constructs with clear semantics to a formal heterogeneous modeling framework which wrap an externally simulated model,
- introducing templates for wrapping the models which are going to be simulated externally, in order to make them co-simulatable,
- proving the concepts using two case studies for three external languages/tools: HDL code running in an HDL simulator, Simulink/MATLAB, and compiled software which is executed/simulated on an embedded soft processor or the host machine.

The main contributions appear in Sec.IV and V, where the modeling framework is extended and templates for wrapping the external models are introduced. However, Sec.II gives an overview of the modeling framework and the formalism which is used later in Sec.IV. Sec.III introduces the co-simulation architecture and Sec.VI validates out approach with two case studies. Finally, related works will be discussed and the paper will be concluded.

II. The Modeling Framework

The ideas presented here are general enough to be applicable to many formal heterogeneous modeling frameworks. ForSyDe (Formal System Design) is the modeling framework which is chosen for the purpose of this work. In this section, we describe how ForSyDe addresses the problem of heterogeneous embedded system design. First, the concept of models of computation is briefly reviewed. Then, we present how a system is specified in ForSyDe as a hierarchical process network, and how the processes in a network are constructed, both intuitively and formally. An exemplified system specification will try to put all the introduced concepts into action.

A. Models of Computation

One important issue is the language that should be chosen for system-level design of an embedded system. Most of today's programming languages aim at providing users with means to concentrate on functional specification of their application by abstracting away low-level details of the underlying implementation technologies. The notion of time is considered as one of these details and has been removed from the semantics of most modern languages[1] [2]. This is very unfortunate for embedded system designers who often need to design real-time systems where the correct behavior is not only defined based on producing the correct outputs, but also on producing them at the right time. The situation is becoming even more complicated due to the introduction of multi- and many-core architectures used in embedded systems design, where both function and execution time are heavily dependent on the synchronization between different cores. In addition, the practical design of different aspects of a heterogeneous system requires different abstractions of time. This calls for a need to support a heterogeneous representation of time in the system design language.

Models of Computation (MoCs) [3] address this problem by defining how different components of a system communicate and interact with each other—taking either physical time or an abstract notion of time into account. For example, in the synchronous MoC, the lifetime of a system is seen as instantaneous reactions, in which each component consumes exactly one data token from all of its inputs and produces exactly one output token on its outputs. The synchronous MoC is suitable for specifying digital circuits and is used also by synchronous languages [4] like Esterel and Lustre. The perfect synchrony hypothesis assumes that the underlying machine is infinitely fast and that outputs are produced at the same time instant as the inputs are consumed. The synchronous MoC implies a total order among the events that happen in behaviors of a system. The same idea has been applied to other models of computations in ForSyDe. For example, to model an analog component of a system, the continuous-time (CT) MoC becomes handy which assumes a non-discrete consumption and production of tokens in time. A formal definition of some of the MoCs used in ForSyDe can be found in [5].

[1]This is partly due to the fact that the instruction set architecture (ISA) of almost all the processors hide the precise timing of the implementation technology.

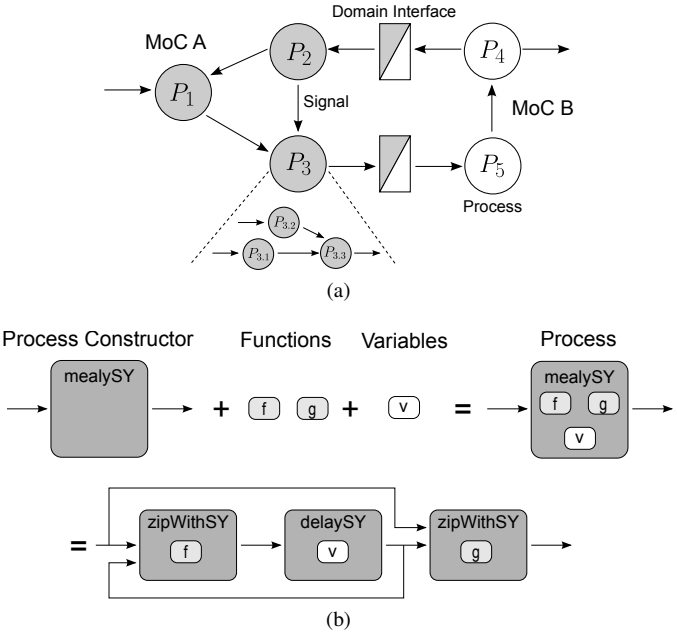


Fig. 1. A Hierarchical Heterogeneous Process Network (HPN) in ForSyDe (a). Using a Process Constructor to build a Mealy machine process (b).

A key concept in ForSyDe, which makes it different from many other modeling frameworks, is that the designer is restricted to use a set of pre-defined constructs in order to specify a system. This restriction leads to a formal model which improves analyzability. This property is typically used in tasks such as automated synthesis, verification, or performance analysis. For example, if a system is modeled using predefined constructs in the synchronous data-flow (SDF) MoC of ForSyDe, the resulting model can be statically scheduled to be implemented on a processor without an operating system with minimum memory requirements [6].

B. System Specification in ForSyDe

In the ForSyDe design methodology, a system is specified as a *Hierarchical (concurrent) Process Network* (HPN). Each process belongs to a single MoC. Processes communicate with each other via *signals* and each process belongs to a specific MoC. Processes which belong to different MoCs are connected via special processes, called *Domain Interfaces* (DIs). Hierarchy is achieved by process composition. Fig.1a illustrates the structure of a hierarchical heterogeneous ForSyDe process network. Here, the process P_3 is made out of three processes $P_{3.1}$, $P_{3.2}$, and $P_{3.3}$.

Functional processes are created using *process constructors* in ForSyDe. Process constructors are in charge of building processes out of user-defined functions and constants, which define the behavior of the resulting process. Fig.1b illustrates this idea for a process that models a Mealy finite-state machine. In order to create a process P that implements a Mealy finite-state machine in the synchronous model of computation, the designer uses the process constructor $mealySY$, and

supplies the function f calculating the next state, the function g calculating the output, and the value v that gives the initial state of the finite state machine. A process P of type $mealySY$ can be created with

$$p = mealySY(f, g, v) \tag{1}$$

In the ForSyDe methodology, functions passed to process constructors must be side-effect-free, which implies that they do not have access to any global value and can not have local static values either. Process constructors in each MoC define the communication and synchronization interface of each process with respect to other processes in the HPN. Intuitively, a process receives input tokens from other processes (or from the inputs of the system), invokes the appropriate functions passed to it, and communicate the output tokens to other processes (or outputs of the system). These behaviors are completely defined by the process constructors used in creating a process. Also, as mentioned previously, the use of process constructors leads to a structured and formal model, which simplifies application of formal methods. The concept of process constructors in ForSyDe has been used to develop a compiler for hardware-synthesis. Earlier work has identified the potential for design transformation [1] through equational transformations based on short cut fusion, since process constructors are higher-order functions.

Signals are used as a communication and synchronization mechanism between processes. In ForSyDe, we follow a similar approach to the tagged signal model [3], where a signal is a set of events, and events are composed of tags (which could be implicit) and values. From the designer's point of view, signals can be thought of as streams of events conveying data tokens between processes.

C. Formalization

First we introduce the concept of signal partitioning as a tool for our formalism. Based on that, the process constructors are defined. For a more precise definition of concepts introduced in this section, please refer to [5].

1) Signal Partitioning: We use the partitioning of signals into sub-sequences to define the portions of a signal that is consumed or emitted by a process in each evaluation cycle. Signal partitioning is more useful when defining other MoCs such as SDF. But, we do not simplify it away here for the synchronous MoC because we use it in defining wrapper processes. A *partition* $\pi(c, s)$ of a signal s defines an ordered set of signals, $\langle r_i \rangle$, which, when concatenated together, form "almost" the original signal s. The constant $c \in \mathbb{N}_0$ defines the lengths of elements in the partition.

For finite signals, a partition of a signal s constructs a finite number of sub-signals and it may drop some elements at the end. Therefore, it is not always possible to completely reconstruct the original signal from the sub-signals defined by $\pi(c, s)$. We usually write $\pi(c, s) = \langle r_i \rangle$ with $i = 0, 1, 2, ...$ to designate the individual sub-signals of the partitioning with r_i. The *remainder* $\mathsf{rem}(\pi, c, s)$ of a partitioned signal consists of the remaining elements of a signal which are dropped by the partitioning. Therefore,

$$s = (\bigoplus_{\langle r_i \rangle = \pi(c,s)} r_i) \oplus \mathsf{rem}(\pi, c, s), s \in S, i \in \mathbb{N}_0 \tag{2}$$

where $\oplus$ is the concatenation operator for signals.

Examples: Let $s_1 = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 \rangle$ and $c_1 = 3$. Then, we get the partition $\pi(c_1, s_1) = \langle \langle 1, 2, 3 \rangle, \langle 4, 5, 6 \rangle, \langle 7, 8, 9 \rangle \rangle$ and the remainder would be $\mathsf{rem}(\pi, c_1, s_1) = \langle 10, 11 \rangle$.

Alternatively, let $s_2 = \langle 1, 2, 3, .. \rangle$ be the infinite signal with ascending integers and $c_2 = 2$. The resulting partition is infinite: $\pi(c_2, s_2) = \langle \langle 1, 2 \rangle, \langle 3, 4 \rangle, ... \rangle$.

2) Process Constructors: A $combSY_m$ process constructor creates a combinational process which takes m input signals, generates one output signal and has no internal state as illustrated in Fig.2. Such a process will continuously get one value from each of its inputs and supplies them to a function to calculate the output. It is defined as:

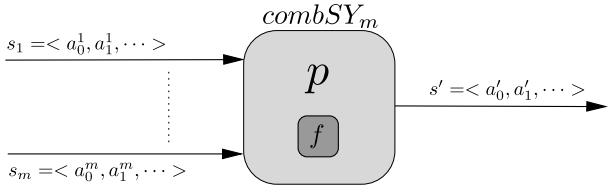


Fig. 2. The process constructor $combSY_m$ creates a combinational synchronous process.

$$\begin{aligned} &combSY_m(f) = p \in P_{SY} \\ &\quad \text{where } p(s_1, \cdots, s_m) = s' \\ &\qquad f(a_i^1, \cdots, a_i^m) = a_i', \mathsf{length}(a_i') = 1, \\ &\qquad \pi(1, s_j) = \langle a_i^j \rangle, \\ &\qquad \pi(1, s') = \langle a_i' \rangle, \mathsf{rem}(\pi, 1, s') = \langle \rangle, \\ &\qquad s_j, s', a_i, a_i' \in S, i \in \mathbb{N},\ 1 \le j \le m \end{aligned} \tag{3}$$

$combSY_m$ is a stateless process constructor, which takes the function f as argument and returns the process p. p in turn is a process with m input signals $(s_1, \cdots, s_j, \cdots, s_m)$ and one output signal (s'). The partitioning of input and output signals state that exactly one token is consumed from each input by the process during each evaluation cycle and exactly one output token is produced. The partitioned signals s_j are denoted by the sequence of sub-sequences a_i^j; the partitioned output signal s' is denoted by the sequence a_i'. The length of each a_i^j and a_i' is 1.

The function f defines the functionality of the process. It takes one input event from signals s_j as arguments and produces one output event. Thus, the process p is defined by f, which is repeatedly applied on parts of the input signals and produces parts of the output signal. We call an application of f an *activation cycle*, *firing cycle* or *evaluation cycle* of the process.

The $delaySY$ process constructor introduces state to a model by making processes that add a delay to the signals. It acts much like a register in digital hardware design.

$$\begin{aligned} delaySY(c) &= p \in P_{SY} \\ \text{where } p(s) &= s' \\ s' &= \langle c \rangle \oplus s \\ s, s' &\in S \end{aligned} \tag{4}$$

The above definition simply says that the produced process concatenates an initial value to the beginning of the input signal. More complex stateful process constructors—such as Moore and Mealy state machines—can be defined based on $delaySY$ and $combSY_m$ process constructors.

D. Execution Semantics

ForSyDe process networks are very similar to datafow process networks which are surveyed in [7]. A closer look at a process constructor like the one defined in definition (3), reveals that signal partitioning implies non-strictness property for the processes. It is only needed to to have a single token (in the synchronous MoC) of the input signals available in order to be able to compute a single output token. This property is called *monotonicity*. Moreover, according to the same definitions in this work, ForSyDe processes are functional processes that are also *continuous* and *sequential*. A sequential process requires tokens from all of its inputs to be available in order to fire and produce the output tokens.

The reference implementation of ForSyDe in the pure functional language Haskell [8] implements processes operating on signals as functions which recursively operating on (infinite) lists. At each invocation, the process tries to fetch the required inputs from the head of the input lists (signals, streams) and calculates the output. Since Haskell has a lazy evaluation style, if the required input token is not available yet, it blocks the execution of the current process (which is a Haskell function) and fires the processes that will produce the required token. This implements a blocking, demand-driven synchronization among the processes in a process network.

Haskell is an advanced, open source functional language with useful features such as type-safety, polymorphic functions, type inference, and lazy evaluation. However, the concepts used by ForSyDe are language independent. For example, ForSyDe ideas are currently being re-implemented on top of SystemC class library in the framework of the European Artemis project SYSMODEL [9].

The SystemC implementation of ForSyDe uses FIFOs with blocking reads and writes in order to implement the same behavior on top of the SystemC scheduler. There, SystemC processes are used instead for ForSyDe processes and sufficiently large (but not virtually infinite) FIFOs can act as signals. In order to preserve properties such as continuity and side-effect-freeness, additional templates and modeling guidelines are provided with the SystemC implementation. The Haskell implementation imposes all the required restrictions at language level and is preferred in this sense. But, SystemC is becoming more popular and is being used more in industry.

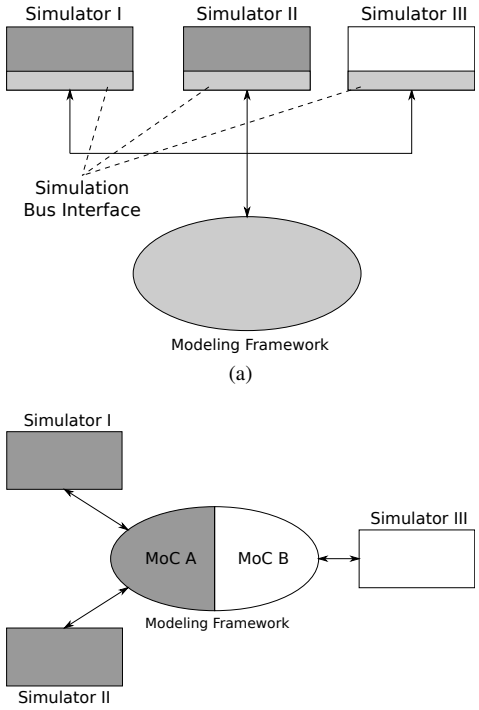


Fig. 3. Heterogeneous co-simulation using a standard interface (a) and using a heterogeneous modeling framework (b).

III. The Co-simulation Architecture

The challenge is to simulate a heterogeneous system with components that are modeled and executed in different tools and languages. Additionally, it is important to know how shall the simulators and the main simulation engine be synchronize together.

A. Integrating Simulators

One way of integrating different simulators is to design a standard interface and require all of the tool vendors to implement this interface. Then, all the tools need to be connected to a so called simulation bus by which the main framework requests simulation services from the tools. In this approach, we are essentially forcing the vendors to implement a specific MoC in their interface—no matter what MoC best describes their models. This situation is depicted in Fig.3a.

Another approach is to let the tools communicate with the main modeling framework using the semantics of their natural MoC and handle the heterogeneity inside the framework. One obvious requirement for the desired design framework is that it should be able to capture the intrinsic heterogeneity of these systems by supporting different levels of abstraction that is assumed for modeling each component. Different levels of abstractions can be captured using various models of computation. In other words, the first step is to understand the MoC which best describes the level of abstraction that a language/modeling tool assumes. Then, a framework which can execute models with multiple MoCs should be chosen and

extended to communicate with other tools and co-simulate/co-execute them together. Fig.3b represents this idea pictorially. Here, Moc A best describes the abstraction level of simulators I and II and MoC B is more suitable to capture the semantics of simulator III. They communicate with the main framework with (almost) the semantics of the MoC that they belong to and the modeling framework handles the communication between the MoCs internally—possibly using domain interfaces.

Based on what has been described in Sec.II, ForSyDe is perfect candidate for such a framework. The core idea is to introduce a new class of processes to ForSyDe which "wrap" an external model inside a selected MoC. These *wrappers* are back-doors in the framework that communicate data and synchronize with external simulators, but appear as ordinary processes in the ForSyDe process network—complying with the semantics of the MoCs to which they belong.

B. Communication and Synchronization

In Sec.II we have seen that ForSyDe processes have no means of synchronization or communication except the data passing through signals.

As the first option, we have exploited the same idea in order to implement the interface between ForSyDe and each simulator. A FIFO-like communication medium with blocking read and write semantics is sufficient for this purpose. Formalizing such a behavior is easy and is in line with what has been done for ordinary process constructors.

However, not all simulators can be controlled in this way. We may need to explicitly control the target simulator engine for each execution step in order to advance the simulation. We do this by issuing a set (or a stream) of commands to the simulator in addition to communicating the input and output data.

IV. Extending ForSyDe for Co-simulation

In this section we will try to formalize an extension to ForSyDe which allows construction of wrapper processes that are used for co-simulation. This formalization eases reasoning about the co-simulation models and makes room for automating tasks such as interface and wrapper generation.

A. Defining the wrapper process constructors

The wrapper process constructor is illustrated in Fig.4 and is defined as:

$$
\begin{aligned}
&wrapSY_m(\Psi) = p \in P_{SY} \\
&\quad \text{where } p(s_1, \cdots, s_m) = (s') \\
&\qquad \psi_i(a_i^1, \cdots, a_i^m) \rightsquigarrow a_i', \mathsf{length}(a_i') = 1, \\
&\qquad \pi(1, s_j) = \langle a_i^j \rangle, \\
&\qquad \pi(1, s') = \langle a_i' \rangle, \mathsf{rem}(\pi, 1, s') = \langle\rangle, \\
&\qquad u_j \equiv \langle a_i^j \rangle, u' \equiv \langle a_i' \rangle, \\
&\qquad s_j, s', a_i, a_i' \in S, i \in \mathbb{N},\ 1 \leq j \leq m
\end{aligned}
\tag{5}
$$

The process constructor accepts Ψ, a sequence of simulation functions $\langle \psi_0, \psi_1, ... \rangle$ and produces a process p. The process p, in addition to its normal input and output signals, conducts an input and a set of output internal streams which are used by the process for communication with the simulator. The communication streams u_j and u' are equivalent to the partitioned input and output signals and could be implemented as TCP queues, named pipes, shared memory communication, etc. The real shape of Ψ is context dependent and can be a set of commands for controlling the simulator engine or even totally absent (in case of synchronization via blocking communication). The $\rightsquigarrow$ operator in the above definition should be read as "yiels". It says that running the simulator at the i^{th} invocation with function ψ_i, on the stream-equivalent of the inputs a_i^j yields the stream-equivalent of a_i'—considering the current state of the simulator. This also enforces the property of monotonicity for the wrapper processes, disregarding the semantics of the simulators being wrapped. In addition to that, writing the left hand side of the yield operator in function-application style is on purpose and it is a requirement for the model wrapper in the simulator to preserve the sequentiality property.

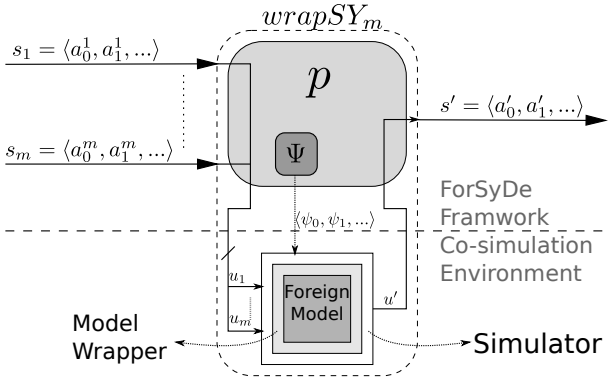


Fig. 4. The process constructor $wrapSY_m$ creates a wrapper process with m inputs and one output in the synchronous MoC, plus the communication and synchronization streams between the simulator and the framework.

Note that in the scenario shown in Fig.4, process p is called the *wrapper process*, the foreign model is called the *wrapped model* and the adaptation layer around it is called the *model wrapper* (to be elaborated more later).

B. Co-simulation Models and Refinement-by-Replacement

A model in which one or more of the processes are constructed using the $wrapSY_m$ process constructor is called a *co-simulation model*.

Unlike the simulation models[2] which are self-contained models, co-simulation models rely on the external simulators to be co-executed with the ForSyDe model. The connections between the ForSyDe simulator and individual simulators should be established via the data and command streams accordingly (see Fig.3b).

Wrappers can be used to follow a bottom-up design for heterogeneous systems, where the implementation of individual modules are known and we need to compose and examine

[2]Originally called specification models in [1].

them as a co-simulation model in a heterogeneous design framework.

Alternatively, if a co-simulation model is obtained by replacing a process in a simulation model with a wrapper process, which wraps a model with a lower abstraction level compared to its original counterpart, a top-down design approach has been followed. We call this *refinement-by-replacement*. Thanks to this new extension, an iterative and partial refinement of a model down to its implementation while keeping the model executable and testable with the original test-benches in each step is possible.

V. Model Wrapper Templates

In Sec.IV a standard interface for the wrapper processes was introduced based on the $wrapSY_m$ process constructor. In this section we will see how the simulated models should interact with the ForSyDe framework by exchanging data and synchronizing with it. This is achieved by introducing *model wrappers* which are adaptation layers around the actual simulated models. Model wrappers depend on the target language/tool which is being simulated and can vary based on the number of input and output data streams. Fortunately, a generic model wrapper template can be introduced for each simulator and it can be even generated automatically by the $wrapSY_m$ process constructor. Here, three popular classes of model wrappers are introduced: a GDB wrapper for wrapping compiled software, an HDL wrapper for digital hardware models, and a Simulink wrapper for high-level models.

A. GDB wrapper

Co-simulating hardware and software together has been studied already for more than a decade. Several techniques on different abstraction levels were proposed and it is still a subject of interest. Here, we are interested in developing a generic model wrapper for compiled embedded software. The software might be running on the designer's host machine (at early development stages where testing the functional correctness is desired), simulated in an Instruction-Set Simulator (ISS), or even run on a real hardware and co-executed with high-level models in a Hardware-In-Loop (HIL) fashion.

The GNU Debugger (GDB) [10] is the debugger provided by the GCC tool-chain which supports a wide range of processors. GDB allows the programmer to run a software, stop it at specified breakpoints, inspect its progress and even modify its environment. Many instruction-set simulators also provide GDB support, which makes it a good candidate for the purpose of this work.

No matter if we are running the software on the host machine, simulating it in an ISS, or running it on actual hardware, we call the GDB instance itself the simulator. The term simulator here is compatible with the terminology we used before (refer to Fig.3b). The wrapper process in ForSyDe can connect to the GDB instance via a TCP connection or a serial line, synchronize, and communicate data with it using GDB commands.

Require: wrapped software model as function f
Require: breakpoints set at lines 2 and 4 during initialization
1: **loop**
2: $inps \leftarrow$ {set by GDB for m inputs}
3: $out \leftarrow f(inps)$
4: $out \hookrightarrow$ {read by GDB}
5: **end loop**

Fig. 5. A generic software model wrapper which communicates with a $wrapSY_m$-based process. It will run in GNU debugger and the wrapper processes communicates and synchronizes with it using GDB commands.

In Fig.5, the pseudo-code for the model wrapper is provided. The model wrapper in this case includes synchronization breakpoints at the beginning and at the end of a loop (lines 2 and 4) where the data is written to and read from memory locations corresponding to input and output variables, using GDB commands. Thus, in this case communication is performed by manipulating the memory locations in the software's address space. The body of the loop only invokes the wrapped software function(s). This does not require the wrapped software to be a stateless function. It could be an infinite loop (more common for control-oriented applications) where it constantly interacts with its environment. In this case the synchronization break points can be set at the positions where the data communication is happening. Note that if the wrapped model requires different amount of data on each invocation, then the synchronous MoC is not a good fit for the wrapper and we should use probably a similarly-defined wrapper in the untimed MoC.

In the GDB wrapper solution, the ForSyDe wrapper process explicitly controls the simulation engine by issuing the required GDB commands. Based on our formalism, the simulation function set can be described as:

$$\begin{aligned} &\Psi_{sw} = \langle \psi_{sw}, \psi_{sw}, ... \rangle \\ &\quad \text{where } \psi_{sw} = \langle \mathsf{set}, \mathsf{continue}, \mathsf{read}, \mathsf{continue} \rangle \end{aligned} \tag{6}$$

which means for each evaluation cycle the simulator (GDB) should:

1) set the input variables with corresponding input data,
2) continue the execution until the next breakpoint,
3) read the results back from output variables,
4) continue the execution until the next breakpoint.

B. HDL Models

As stated before, we have restricted this work to the synchronous model of computation. So, the intension here is to wrap a model in the synchronous subset of hardware description languages. The synthesizable subset of VHDL and Verilog HDL fall into this category. The assumption here is that the model is being simulated using an HDL simulator. The model wrapper appears as a top-level module which instantiated the wrapped model and communicates input and output data from/to the ForSyDe framework to/from it.

According to the previous section, there are at least two ways to synchronize the HDL simulator with the ForSyDe

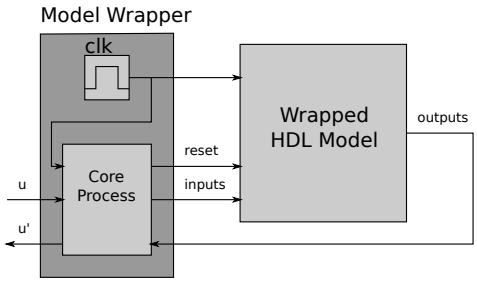


Fig. 6. Realization of an HDL model wrapper.

framework. Either control the HDL simulator engine from the framework, or synchronize using the input and output streams. The first approach can be implemented using the TCL command interface which some commercial simulators such as Modelsim provide. The second solution needs more general primitives for realization and can be implemented using file I/O and Unix named pipes and is the one chosen for this work. Named pipes in the Unix-based systems appear as simple files on the file system and have a FIFO with blocking read and write semantics. Using standard facilities in HDLs such as the `TEXTIO` package in VHDL, the model wrapper can write to and read from them. In this case, although the simulator is not explicitly controlled using simulation commands, it can not run at any arbitrary speed because the read and writes to Unix pipes act as breakpoints.

One more issue to be taken care of is generation of clock and reset inputs for the wrapped model. In the synchronous MoC of ForSyDe, no explicit clock exists. If the wrapped model contains registers, then the model wrapper is needed to generate these signals and feed the HDL model with them. A valid periodic clock must be generated in the model wrapper which will be used both by the core process of the wrapper and also fed to the wrapped model. Obviously, all the model wrappers in the same MoC should generate the same clock signal which complies to the semantics of the implicit clock used in the SY MoC. But there is no need to explicitly connect the clock signals of different model wrappers and keep them synced (unless enforced by external constraints).

The structure of an HDL model wrapper and its interaction with the wrapped HDL model is illustrated in Fig.6. The pseudo-code in Fig.7 shows how the core process of such a model wrapper can be realized. It is important to note that data-type translation may be needed for a successful data communication. For simple data-types, writing and then reading them back as strings may be sufficient, but for more complex data structures such as vectors and records, an additional serialization before writing and deserialization after reading is required.

Since there is no explicit communication between the wrapper process and the simulator engine, there is no need for the simulator functions and therefore they are absent.

$$\Psi_{HDL} = \langle \perp, \perp, ... \rangle \tag{7}$$

where $\perp$ denotes absence of value in the stream.

Require: a periodic clock clk
Require: input and output pipes $ipipe$, $opipe$
1: $reset \leftarrow active$
2: wait until $rising\_edge(clk)$
3: $reset \leftarrow deactive$
4: **loop**
5: $inps \leftarrow read(ipipe)$ {for m inputs}
6: **if** $inps = \emptyset$ **then** {end of simulation}
7: exit loop
8: **end if**
9: $model\_inputs \leftarrow inps$
10: wait until $rising\_edge(clk)$
11: $out \leftarrow model\_outputs$ {packed to a single output}
12: $write(opipe, outs)$
13: **end loop**

Fig. 7. The core process of a generic HDL model wrapper which communicates with a $wrapSY_m$-based process. It assumes an arbitrary valid clock is also generated in the model wrapper.

C. Simulink Models

Simulink is a graphical modeling environment with a customizable set of block libraries that are used to model and simulate embedded systems. Its solver is able to simulate systems in the continuous time domain, but can be parameterized to execute optimally also for discrete-state systems. Here, the aim is to wrap a discrete-time Simulink model in the ForSyDe framework. Communication and synchronization is performed in the same way as HDL wrappers. No direct interaction with the simulator but synchronization via blocking reads and writes to named pipes. This is achieved by introducing two new blocks to the Simulink library: an input source and an output sink block. The top level Simulink model (wrapped model) is connected to these blocks which together form the model wrapper. Both blocks open the files (pipes) on simulation start and close them on exit. The sink block simply writes the received sample to the output pipe upon each invocation. The source block reads the input samples from the input pipe and returns them as a vector in the block output.

As like the HDL wrapper, the simulation functions are absent due to absence of the need for explicit synchronization.

$$\Psi_{Simulink} = \langle \perp, \perp, ... \rangle \tag{8}$$

VI. Case Studies

We demonstrate the usage of our proposed co-simulation method using two case studies. The first example is modeled in synchronous MoC and includes two wrapper processes in order to co-simulate the main model with external tools running components in different levels of abstraction. In the second example, the model uses several MoCs, where only the synchronous part of it is implemented is simulated/executed externally with the main model.

A. An Audio Equalizer

The main task of the equalizer system is to adjust an audio signal according to the user input. In addition, the

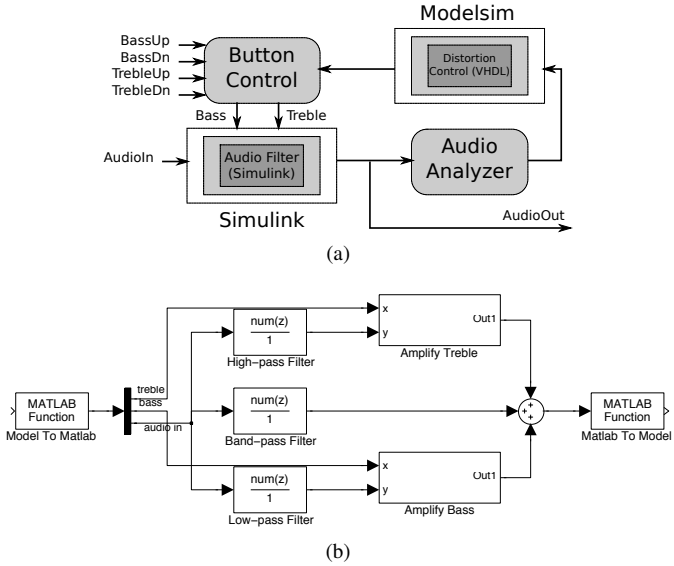


Fig. 8. The top-level block-diagram of the equalizer example (a). The wrapped model and the model wrapper blocks for `AudioFilter` module in Simulink (b).

bass level must not exceed a pre-defined threshold to avoid damage to the speakers. The `ButtonControl` subsystem monitors the button inputs and the override signal from the subsystem `DistortionControl` and adjusts the current bass and treble levels. This information is passed to the subsystem `AudioFilter` , which receives the audio input, filters and amplifies the audio signal according to the current bass and treble levels. This signal, the output signal of the equalizer, is analyzed by the `AudioAnalyzer` subsystem, which determines, whether the bass exceeds a pre-defined threshold. The result of this analysis is passed to the subsystem `DistortionControl`, which decides, if a minor or major violation is encountered and issues the necessary commands to the `ButtonControl` subsystem. Fig.8a illustrates the top level of the equalizer system.

The `AudioFilter` module, which is mainly a dataflow model, is modeled in Simulink and is wrapped as a process in ForSyDe. Filters are modeled in Simulink because Matlab/Simulink provide a good support for designing signal processing blocks and allow us to directly implement the designed blocks. Fig.8b shows the Simulink implementation of this module. Also, a VHDL implementation of the `DistortionControl` module, which is mainly a Mealy state machine, is generated by the ForSyDe compiler from a deep-embedded specification. The VHDL code is run in the Modelsim simulator and is also wrapped as a process to be co-simulated with the rest of the system.

The overall setup, which is composed of the ForSyDe simulation, the MATLAB/Simulink engine and the Modelsim instance are co-simulated and produce the desired results.

B. A Transceiver System

The transceiver system (shown in Fig.9a) is composed of a transmitter, a receiver, and a model of environment in which the radio signal is being transmitted.

The transmitter encrypts its input data, modulates it using the Amplitude-Shift Key (ASK) method and amplifies it before sending it to the antenna. The environment model adds a Gaussian noise to the signal and attenuates it. In the receiver, the signal is demodulated, and decrypted. The output amplification gain of the transmitter is adaptive and is tuned based on the error observed in the receiver output. This model uses the synchronous, untimed (synchronous data-flow), and continuous-time MoCs of ForSyDe for different components and is verified by simulation (Fig.9b–Fig.9d).

The encryption and decryption modules are implemented in software and wrapped in the synchronous MoC. The encryption algorithm used here is Simplified DES (SDES). This is a typical scenario where the implementation of a component is available as an IP block and we intend to integrate it with our system model.

The same wrapper has been used for co-simulating with a GDB instance which is executing the software in the following cases:

(a) running the compiled software on the host Linux machine,
(b) simulating the compiled software in a Nios II ISS,
(c) executing the compiled software on a Nios II processor. In this case, the GDB instance uses a JTAG interface to communicate with an Altera DE2 development board. The Nios II soft processor is implemented on a Cyclone II FPGA on this board and runs the compiled wrapped software.

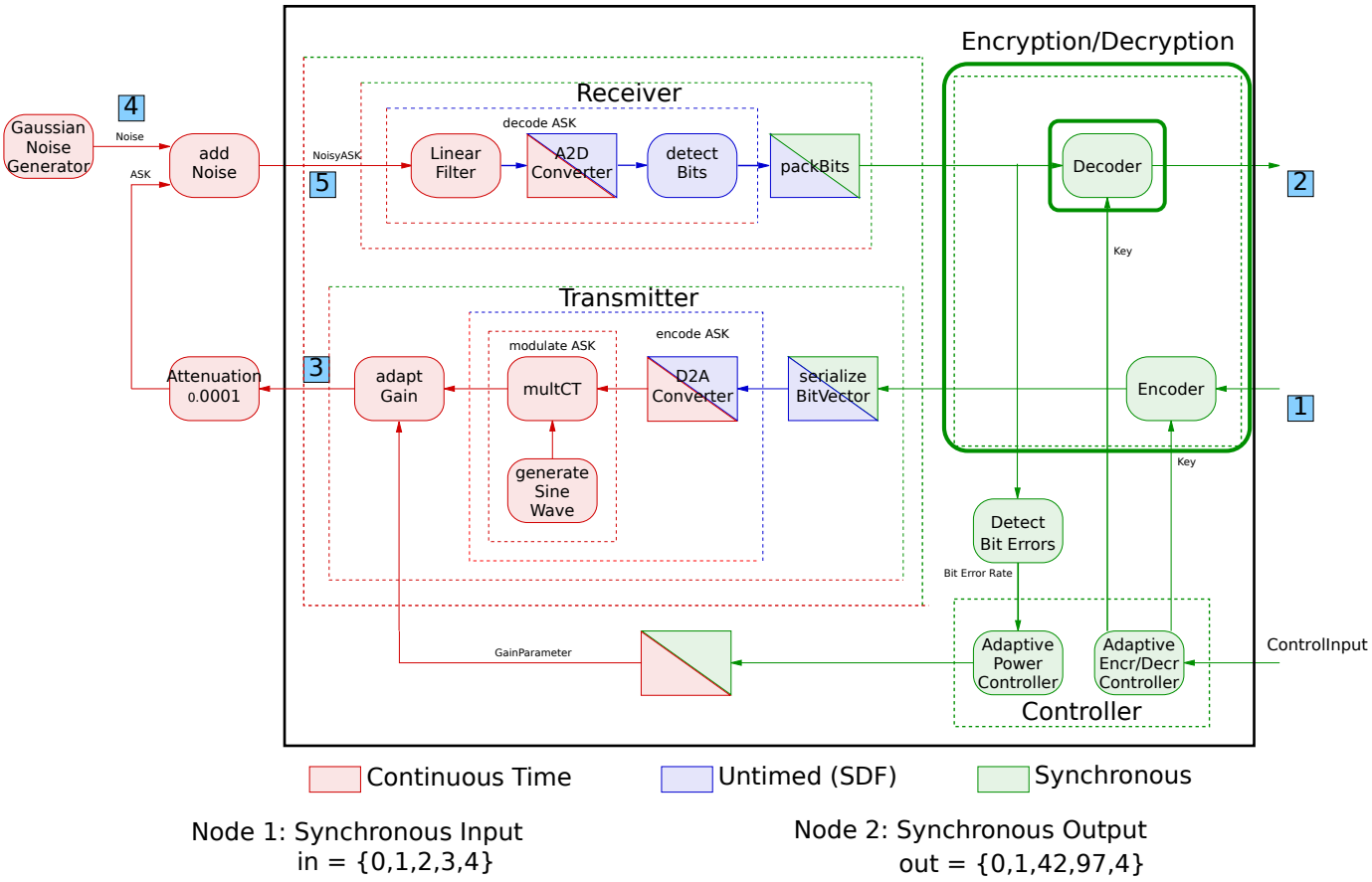
In Fig.9a, the output result of the simulated system together with the waveform of selected intermediate waveforms are illustrated. TABLEI reports the observed execution times for different simulation set-ups.

TABLE I
EXECUTION TIMES FOR SIMULATION OF THE TRANSCEIVER EXAMPLE.

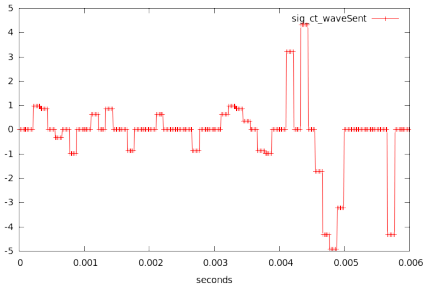
Simulation Set-up	Execution time (s)
Pure ForSyDe Model	0.97
Co-simulation with ISS	5.0
Co-execution with Nios II	9.8

VII. RELATED WORK

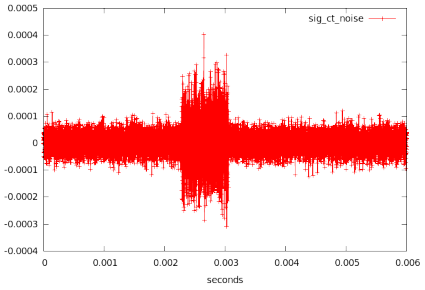
There exist other modeling frameworks that support heterogeneous system modeling based on the concept of models of computation. A notable example is Ptolemy [11] which supports a rich set of MoCs, but takes a different, actor-based implementation approach. Unlike ForSyDe, heterogeneity in Ptolemy is restricted to hierarchical composition in favor of a more disciplined approach and actors in the same level of hierarchy should belong to the same MoC. This restriction sometimes become annoying, especially for top-level models where heterogeneous components need to be composed. HetSC [12] is based on System-C and implements some MoCs on top of the SystemC discrete-event simulation engine. In



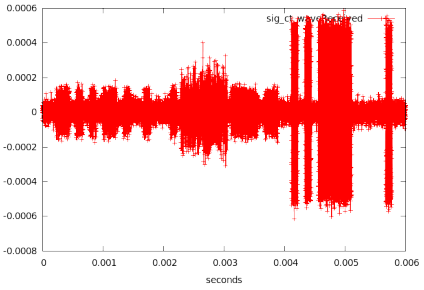
(a)



(b) Node 3: Transmitted wave



(c) Node 4: Added noise



(d) Node 5: Received wave

Fig. 9. The transceiver system uses synchronous, SDF and continuous-time MoCs. The simulation is run with a sample input and the output is shown (a). (c) shows the added noise in the environment while (b) and (d) show the transmitted and recived signals. The synchronous encryption/decription module is implemented in software (C code) and wrapped as a process in ForSyDe.

contrast to ForSyDe, HetSC does not claim a solid formal basis.

A considerable amount of work has been dedicated to hardware-software co-simulation. In [13, Sec.3] a survey of the well-known techniques is given. There, the main attempt is to boost the co-simulation speed via ad-hoc methods rather than providing a general solution for heterogeneous modeling.

Transaction-Level Modeling (TLM) is also a popular approach for system-level modeling and simulation nowadays. TLM is not a well defined, single abstraction level. Instead it is usually defined as abstraction levels above RTL. Depending on the amount of timing details in the model, TLM is divided into Programmer's View (PV), Programmer's View-Timed (PVT), etc. But, models in none of these abstraction levels have clearly defined semantics. Therefore, the main usage of TLM is restricted only to achieving faster simulation speeds for prototyping.

Commercial tools such as MATLAB/Simulink support integration with external tools and even implemented models using features such as Processor-in-Loop or Software-in-Loop. The drawback is that although Simulink is used in many application areas, it is not by itself a heterogeneous modeling environment. Also, we were unable to find a formal description of the semantics of Simulink models.

VIII. Conclusion

We have presented the concept of wrapper processes in the context of ForSyDe—a formal heterogeneous modeling environment for embedded system design. They allow the designers to integrate legacy codes and models in a formal modeling framework. Each wrapper processes lives in a Model of Computation (MoC) and ForSyDe supports integration of several MoCs with different abstractions of time. ForSyDe wrappers are general enough to be able to make the co-simulation with any other simulator possible. This generality, does not call for any special support from tool vendors apart from simple input/output operations that they usually provide. To wrap a model, an appropriate model of computation which best approximates the semantics of the simulator should be selected and then a wrapper process should be instantiated in the corresponding MoC in ForSyDe. A set of templates, called model wrappers, were presented in order to make the co-simulated models able to communicate and synchronize with the ForSyDe wrapper processes. Model wrappers can be generated mostly automatically based on the introduced templates.

As like other processes in ForSyDe, a process constructor has been formally defined to separate a ForSyDe wrapper process functionality from its communication and synchronization semantics. This formalism helps us to develop automated methods for generation of co-simulation models from simulation models and refinement of a specification model to its implementation. Wrappers can be used in a top-down design approach with the refinement-by-replacement method. We will investigate this more as an extension to a design flow in our future work. Although in this work we have shown implementation examples for wrappers in the synchronous MoC, extending them to other MoCs such as the untimed and discrete-event MoCs should be easy and straightforward. Defining process constructors and introducing model wrapper templates for the continuous-time MoC needs further investigation and together with the rest will be an extension of this work in the future.

Acknowledgment

This work is partially supported by SYSMODEL project [9] in the framework of the European ARTEMIS program.

References

[1] I. Sander and A. Jantsch, "System modeling and transformational design refinement in ForSyDe," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 1, pp. 17–32, January 2004.

[2] E. A. Lee, "Computing needs time," *Commun. ACM*, vol. 52, no. 5, pp. 70–79, 2009.

[3] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 12, pp. 1217–1229, December 1998.

[4] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. D. Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, January 2003.

[5] A. Jantsch, *Modeling Embedded Systems and SoCs*. Morgan Kaufmann, 2004.

[6] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[7] E. Lee and T. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[8] The Haskell programming language home page. [Online]. Available: http://www.haskell.org

[9] The SYSMODEL project home page. [Online]. Available: http://www.sysmodel.eu

[10] GDB: The GNU project debugger home page. [Online]. Available: http://www.gnu.org/software/gdb/

[11] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity - the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[12] F. Herrera and E. Villar, "A framework for heterogeneous specification and design of electronic embedded systems in SystemC," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 3, pp. 1–31, 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1255456.1255459

[13] M. Loghi, T. Margaria, G. Pravadelli, and B. Steffen, "Dynamic and formal verification of embedded systems: A comparative survey," *International Journal of Parallel Programming*, vol. 33, no. 6, pp. 585–611, 2005. [Online]. Available: http://www.springerlink.com/content/f6gt2010185n6260/